Substitute Specification

TITLE OF THE INVENTION

PROGRAM MONITORING METHOD AND DEVICE

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0001] The invention relates to a method and an arrangement for monitoring a program.

2. Description of the Related Art

[0002] A program is normally monitored by means of specific brake points at which a programmer can view debugging information and, if necessary, can deduce that there is a malfunction in the program. This manual measure is extremely tedious and time-consuming in the case of a sufficiently large program with numerous subroutines. Furthermore, the debugging information is mostly evaluated offline, that is to say not during the time in which the program is running. If the program is part of a distributed system, it is no longer feasible to coordinate clear, manual monitoring.

SUMMARY OF THE INVENTION

[0003] The object of the invention is to specify a method and an arrangement for monitoring a program, which allow efficient monitoring even when the program is running, particularly in a distributed system. The present invention allows such monitoring to be carried out even for a large number of programs.

[0004] In order to achieve the object, a method for monitoring a program is specified, in which the program has an instrumentation part added to it. The instrumentation part generates a message, and transmits this message to a monitoring process. The monitoring process initiates a predetermined action.

[0005] An instrumentation part is a program part or a (program) code (fragment) which is linked to the program to be monitored. This link can be produced by embedding the instrumentation part in the program itself, or in a function associated with the program. By way

of example, the program can itself be instrumented at a number of points, each of which then sends messages to the monitoring process when specific events occur. At the same time, an output routine in the program could be instrumented so that, in addition to the normal (that is to say without any instrumentation) output from the program, further outputs are passed to the monitoring process (instrumentation of the input/output interface of the program).

[0006] The monitoring process preferably coordinates the messages which are sent (possibly by numerous distributed) programs. Furthermore, the monitoring process may have a number of different actions which it can initiate or start as a consequence of the messages:

- For example, the messages can be displayed. The messages are preferably displayed as a function of time, for example in the form of a message sequence chart (MSC). Displays in the form of a tree structure or in the form of lists are also feasible.
- Furthermore, it is possible to intervene in the running of the program. For example, the instrumentation part could cause the program to wait for a response from the monitoring process before itself being continued. This response can be produced by a user/programmer, or can be produced automatically as a function of specific presets (IF-THEN-ELSE structure). The automated running can be carried out as a dedicated subroutine, coordinated by the monitoring process.
- It is also possible for the action to correspond to an open or closed-loop control task. One example of this would be switching a unit in a technical system on or off as soon as specific presets are satisfied or have been signaled to the monitoring process.

[0007] The advantages of the method proposed here include the capability to monitor the program while it is running. The instrumentation ensures that specific events are signaled, even while the program is running, in the form of messages to the monitoring process, where these messages are collected and processed in a suitable manner. In particular, the monitoring process can use a filter function to ensure that the only messages which are displayed or taken into account are those which actually correspond to the filtered presets. Furthermore, the capability for use in a distributed system is a major advantage. The complex interaction between a large number of different programs in a distributed system makes fault tracing difficult. The transmission of messages to the monitoring process from each instrumented (also distributed) program when there are a large number of different programs which do not all run on the same computer platform allow - after suitable filtering and/or processing, for example on the basis of a message sequence chart - the preservation of an overview, thus making it

considerably simpler to trace a specific action in the distributed system, and hence considerably simplify the fault tracing process associated with this.

[0008] In this case, it should be noted that a program as described above may invariably also in each case be a part of a cohesive program or a function linked to (associated with) the program.

[0009] It is particularly advantageous for middleware which is associated with the program to be monitored to be instrumented. In this case, a functionality which is controlled by the program can be instrumented centrally. This is actually a major advantage since, for example, there is no need to instrument all the input/output routines in the program in this way, with only the one associated routine being instrumented in the middleware, instead of this. This actually corresponds to instrumentation of all the input/output routines in the program to be monitored. The actual addition at the one point in the middleware is extremely economic, since there is no need to search through the entire program to be monitored for said routines. The instrumentation can be changed centrally, at one point, for all the affected routines (interface encapsulation). Furthermore, this type of instrumentation is highly flexible and can be added to. If the program to be monitored changes, the instrumentation is already prepared, in terms of the affected routines (in this case, the input/output routines by way of example) for the next program to be monitored.

[0010] In particular, in a distributed system, that is to say in a system which has a number of computers which are linked to one another by a network, the following mechanisms are monitored in the program:

- Message transmission (message passing, task to task):
  A receiver (program or process) waits for a message. It cannot continue until the message has been transmitted by a transmitter.
- Remote procedure call (RPC):
  By way of example, a process tries to start a program on another computer (remote). The process waits until the remote computer has completed the processing. Normally, the process is informed whether the processing has been successful or unsuccessful.
- Task join:
  A process is added to another process. The added process does not exist any longer from then on.

[0011] If the system is distributed and has a number of computers, then each system is intrinsically autonomous and has, for example, its own system clock. In order to ensure semantic correctness, it is important to check the plausibility, for example the time stamp, of the respective messages. Each message that is sent is, in particular, given a time stamp which can be used for synchronization. For example, the transmission of a message may require a specific time period, and the time sequence must be clarified when displaying a number of messages in the monitoring process. If the system clocks in the individual systems become desynchronized (which must generally be assumed to be the case), then the monitoring process defines the correct time sequence of the transmitted messages. This is done, in particular, by means of a heuristic, which checks whether an assumption is or is not correct. If this is the case, the most recently transmitted time sequence is adopted, and if not, a next assumption is checked. The check is based on message semantics: for example, a response cannot occur before a question. This makes it possible to deduce the relationships between the two system clocks involved. A time offset is defined, which satisfies plausibility of the system clocks. The offset can be adapted as soon as a further system clock with a reference to at least one of the two specific system clocks needs to be taken into account.

[0012] The method described above can preferably be used for testing, controlling and maintaining the program, or a technical system associated with the program.

[0013] In order to achieve the object, an arrangement for monitoring a program is also specified, in which a processor unit is provided, which is set up in such a manner that

[0014] a) the program can have an instrumentation part added to it;

[0015] b) the instrumentation part generates a message and transmits it to a monitoring process;

[0016] c) the monitoring process initiates an action.

[0017] This arrangement is particularly suitable for carrying out the method according to the invention, or one of its developments explained above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0018] Exemplary embodiments of the invention will be explained and described in the following text with reference to the drawing, in which:

Figure 1 is a block diagram with logic components for monitoring a program;
Figure 2 is a timing diagram of mechanisms to be monitored in a distributed system; and
Figure 3 is a block diagram of a processor unit.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0019] Figure 1 is a block diagram with logic components for monitoring a program. The program 101 to be monitored is provided with a filter functionality 102 and instrumentation 103. The monitoring process mentioned widely above has blocks comprising an event generator 104, an event manager 109 and the various display options 115 to 120 (views).

[0020] The event generator 104 contains a parser 105, two message interfaces 106, 107 and a file 108. The event manager 109 comprises a merger 110, an event list 111 (trace event list), a filter unit 112 and a filtered list 113. The respective entry within the filtered list 113 is selected via a navigation unit 114 (stepper).

[0021] The filter functionality 102 is provided for selecting specific instrumentation parts of the instrumentation 103 which are of interest for the respective application. The result of the instrumentation, the messages, are transmitted to a communication interface (socket) 107, or are written to a file 108. The incoming messages are processed in the parser 105, and are transmitted to the merger 110, where the entries of the messages received from the parser 105 are stored in a list 111. In particular, the semantic check of the time stamps attached to the messages is also carried out here. Optionally, in block 112, all the messages stored by the merger 110 in the list 111 are filtered on the basis of specific presets. This results in the filtered list 113, whose entries can be selected by means of the stepper 114.

[0022] The stepper 114 is used as the interaction medium for the user. Input can be entered there, and are passed on via the socket 106 of the event generator 104 to the instrumented program part 103. In particular, this interaction allows an interplay between the program and the instrumentation.

[0023] The interaction can take place by means of a user input or, alternatively, via a rule on the basis of "if ... then ...". Furthermore, an action 121 can be carried out which, against the background of the monitoring process, leads to external control or some other external intervention.

[0024] The filtered list 113 can be processed in various representations 115 to 120. The data in the filtered list 113 can be displayed as:

- list (block 115),
- hierarchically sorted view (for example on the basis of computers or the structure of programs and processes, block 116),
- message sequence chart (MSC, block 117), in which case it is possible to take account of predetermined grouping,
- detailed view (each message has an associated range of information relating to a transmitter, receiver, time stamp, message content, etc; block 118),
- list of the user inputs (block 119),
- test report (block 120).

[0025] Figure 2 is a timing diagram of mechanisms to be monitored in a distributed system. To this end, Figure 2 is subdivided into Figures 2A, 2B and 2C. The horizontal line in each case represents the time axis t.

[0026] Figure 2A shows the message transmission. A transmitter 201 sends a message at a time t1, and this is received by a receiver 202 at a time t2. The receiver 203 has already started to wait for this message at a time t3. It waits until the time t2, before the receiver 202 continues with its task.

[0027] Figure 2B shows the remote procedure call (RPC) mechanism. A process 203 sends a "CLIENT_BEGIN(Call object.method(...)" message to an addressee 204 at a time t4. This message arrives at the addressee 204 at a time t5, the addressee 205 preferably being a specific computer, remote from the computer on which the process 203 is running. The addressee 204 initiates the execution of the transmitted command at this time t5 by means of the "SERVER_BEGIN" command. The execution in the addressee 204 is completed at a time t6, the task is stopped ("SERVER_END") and the result "res" is sent back to the process 203

("Return res=obj.meth(...)"). The result arrives at process 203 at a time t7, and the RPC is ended by the "CLIENT_END" command. The process has waited from t4 to t7 and now continues using the result "res" determined in the addressee 204, for its processing.

[0028] Figure 2C shows an example of a control flow. In this case, the intention is to join two processes to one another ("Task Join"). A process Task1 205 sends a "JOIN_REQUEST" command to a process Task2 206 at a time t8. When it arrives there, at a time t9, a "JOIN_START" command results in the process of joining the Task2 206 process to the Task 1 205 process, which lasts until a time t10. At the time t10, the "JOIN_DONE" command results in a return to the process Task1 205, which resumes its task with a "JOIN_END" command at a time t11. The process Task1 205 waited from t8 to t11, and the process Task2 206 is ended after t10.

[0029] Figures 2A, 2B and 2C show various mechanisms which can be monitored in a distributed system in order to obtain an overview of the interaction between programs and processes running in a distributed manner. Particularly for fault tracing or during maintenance work, it is an inestimable advantage to be able to trace actions at the interface between the processes.

[0030] Figure 3 is a block diagram of a processor unit PRZE. The processor unit PRZE has a processor CPU, a memory SPE and an input/output interface IOS, which is used in various ways via an interface IFC: a graphics interface allows an output to be seen on a monitor MON and/or to be output on a printer PRT. Inputs are entered via a mouse MAS or a keyboard TAST. The processor unit PRZE also has a databus BUS, which provides the connection from a memory MEM, the processor CPU and the input/output interface IOS. Furthermore, additional components can be connected to the databus BUS, such as additional memory, a data memory (hard disk) or a scanner.